LHCb-INT-2013-006
March 4, 2013

A New Nightly Build System for LHCb

M. Clemencic[1] and B. Couturier[1].

[1] *CERN, Geneva, Switzerland*

Abstract

The nightly build system used so far by LHCb has been implemented as an extension on system developed by LCG Application Area [1]. Although this version basically works, it has several limitations in terms of extensibility, management and ease of use, so that the SFT group decided to develop a new version based on a commercial continuous integration system.

Since we cannot adopt the new planned SFT nightly build system because of technical reasons, we decided to investigate the possibility of a custom version based on the open source continuous integration system Jenkins [2].

In this note we describe the implementation of a working prototype of the new nightly build system.

1 Introduction

We have been using the Nightly Build System based the the LCG Application Area one for several years [1]. The system works, but it has got limitations that make it very difficult to extend and manage. A couple of years ago, the SFT group had planned a rewrite of their code base, to overcome the limitations, still allowing extensions like the ones needed by LHCb. We decided to wait their new implementation before reviewing our code. Only recently the SFT group opted for an completely different implementation based on a commercial continuous integration solution. We will not be able to extend their new system for our use case without a complete rewrite of our code (and expensive license costs), so we decided to investigate the possibility of a new implementation of the Nightly Build System based on the the open source continuous integration system Jenkins [2].

The outcome of the investigation is the working prototype described in this note.

1.1 LHCb Software in the Nightly Builds

To better understand the following sections, it is useful to get acquainted with some concepts and terms used in the context of the LHCb Software.

LHCb Software is divided in projects (releasable entities) with dependencies between them, meaning that a project uses libraries produced in another project. So, changes in a software project do not only affect the project itself, but also the projects depending on it. To validate the software, then, we need to build a consistent stack (dependency chain) of projects, which, in the Nightly Build terminology, we call a *slot*.

As part of our Quality Assurance policies and for portability, we can build our software on a few Linux OS flavors, using different compilers and different options (mainly optimized and debug). We call the combination of CPU architecture, OS, compiler and flags a *platform*.

To validate the changes of the software in the widest possible range of cases, in our Nightly Builds we build several slots (with different configurations) on several platform. In some cases, we also need to prepare ad-hoc slot configurations for special temporary validation tests.

2 Requirements

For the paste experience we collected a minimal set of functionalities that a Nightly Build System must provide. It must provide the same functionalities of the old one, such as

- build and test several slots on several platforms
- easy configuration of the content of the slots
- separate the builds of different platforms
- allow customized checkouts (i.e. non default versions of the packages)

- run the tests of a project while building the following one on the stack
- configurable parsing of the build logs (ignore some warnings and errors)
- distribute efficiently the load on a pool of build machines
- a dashboard updated incrementally with accessible links to problems and an overview on all the slots

but, wherever possible, improved and with a simpler and more maintainable implementation.

In additions we want to have some long awaited new features:

- monitoring of the status of the builds
- easy restart at different levels: everything, one slot, one platform of one slot
- produce archives of the checkout and of the builds
- easy creation of new slots (both production and testing)
- manageable procedure for the development of the system itself

3 Design

The new Nightly Build System is divided in three main parts to address the concerns: the configuration of Jenkins jobs (for the coordination and distribution of the tasks), the core tools (for the actual checkout and build tasks), the dashboard (summarized presentation of the results of the builds).

Although Jenkins allows arbitrary complex scripts in the build steps, it is suggested in the documentation to keep the build steps simple, wrapping the complexity of the build in tools that are distributed and developed together with the project. The main reasons are that the web interface provides only a minimal text field (not suitable for development) and that it is not possible to keep track of the evolution of the code of a configuration with a version control system.

In our case, the scripts used for the heavy-lifting part of the builds are hosted on a dedicated GIT [3] repository, instead of living withing the software projects, because they are generic and apply to whole software stacks, i.e. interdependent sets of projects.

An important difference in the design of the new system with respect to the old one is that the various actions required in the nightly builds (checkout, build, test, etc.) are performed by dedicated independent scripts instead of being phases of a monolithic script. Thus it is possible to develop and test a single action without having to restart the whole process from scratch. Moreover, the core tools are meant to work and produce files in any directory, instead of using fixed locations as in the old system, simplifying furthermore the development. Of course, whenever feasible, common code is factored out and shared between all the scripts.

The configuration of Jenkins required the installation of several plug-ins on top of a vanilla installation of the application. The jobs, in Jenkins terms, configured are of two main categories: jobs representing the nightly build slots and generic jobs for the individual steps.

The dashboard is still under investigation. The two main options investigated are CDash [4] and the dashboard of the old system. It is also possible to use something integrated with Jenkins or a completely new custom dashboard. The details will be discussed in a dedicated section.

4 Implementation

4.1 Core Tools

The core tools are hosted on the GIT repository[1]

`http://cern.ch/lhcbproject/GIT/LHCbNightlies2.git`

In the following sections we describe in some details the main components of the Core Tools and their implementations.

4.1.1 Configuration

The old nightly build system uses an XML-based configuration describing in one file the slots to be built, their content, the platforms they should be built for etc. While prototyping the new system it seems reasonable to review the layout and the details of the old configuration file to see what could be simplified or removed.

Because the new system is more modular than the old one and the management part is separated from the build part, we started from a minimal configuration file describing a single slot. To simplify the prototyping phase, we have chosen the JSON format (semi-structured) instead of XML (structured), and, because of the way JSON objects are converted in Python, inside the code we used simple nested Python dictionaries. The format and the internal representation of the configuration are still under discussion and they will probably change in a production version of the system to include, for example, a validation mechanism.

The configuration of a slot consists of a JSON object with the following format:

slot name of the slot (mandatory)

projects list of objects describing the projects in the slot, with each object containing the fields:

[1]LHCb Core Software Team set up a minimal GIT hosting service described in a TWiki page:

`https://twiki.cern.ch/twiki/bin/view/LHCb/GitRepositories`

It must be noted that these repositories will be migrated to the CERN-IT GIT hosting service as soon as available.

  - **name** name of the project (mandatory)
  - **version** baseline version of the project (mandatory)
  - **dependencies** list of project names within the slot that the project depends on (mandatory)
  - **overrides** object containing a simple mapping between package name and required version as a string or *null* if the package should be removed (optional)
  - **checkout** name of the checkout function (optional)

warning_exceptions list of regular expressions matching warnings that should be ignored (optional)

error_exceptions list of regular expressions matching errors that should be ignored (optional)

env list of definitions of environment variables as strings in the format "`<name>=<value>`" (optional)

preconditions list of objects describing a function call as the function name and the arguments of the call (optional)

USE_CMT boolean telling of the slot should be build with CMT rather than the default CMake (optional)

cmake_cache mapping key–value defining variables to pre-load in the CMake cache to tune the build (optional)

Fields declared as *mandatory* in the above list are required in at least one step of the nightly build system and cannot have a default value, but some of them are not used in all the steps, so can be considered optional when testing only one step of the system. The various sections of the configuration file are described in the following sections, including which field are actually used by the step.

For people used to the old configuration, it might seem that the new configuration is more limited than the old one, but it is mainly an impression due to the simplifications introduced. For example, the old option file used different XML tags to add a package to a project or to change the required version, but in this simplified configuration it is enough to declare the version of a package to add it or to change it.

It should be noted that the list of platforms, the declaration of special directories or URLs is not included in this option format. The reason is that those informations are not needed to checkout and build a slot, but are part of the management of the nightly build system, which is responsibility of the Jenkins part of the new system.

The field *env* is used by some of the steps to set environment variables before performing the actual tasks. It is meant to replace the XML tags *cmtprojectpath* and *cmtextratags* of the old configuration with a simpler an more generic option. It must be noted that the placeholders `%DAY%`, `%YESTERDAY%` and `%PLATFORM%` in the old configuration are replaced in the new configuration by the environment variables, respectively, `${TODAY}`, `${YESTERDAY}` (set internally) and `${CMTCONFIG}` (taken from the inherited environment).

To allow for a smooth transition from the old system to the new one and for testing the new system in parallel with the old one, we introduced a simple layer that extracts the configuration of a slot from the XML configuration and produces a dictionary with the same format than the one obtained from the new JSON format.

In order to simplify the development and testing in the new system, the configuration file must be passed as command line argument to each script. The translation layer than converts from XML is automatically triggered when the file name passed on the command line contains the suffix `#slotname` which specifies the name of the slot that should be extracted from the XML[2].

4.1.2 Checkout

The new system features a dedicated script (`StackCheckout.py`) for the checkout of all the projects in a slot. It uses only the fields *slot* and *projects* from the configuration file, and for each project only *name*, *version*, *overrides* and *checkout*.

The aim of the script is to check out all the projects to be built in the slot, applying the required *overrides* and fixing the interdependency declarations.

For each project it is possible to choose a checkout function passing its name[3] as the value of the field *checkout*. Few basic checkout functions are already provided:

defaultCheckout (default choice, if not specified) use the `getpack` command

noCheckout do not checkout the project

specialGaudiCheckout test checkout function used to test the build of Gaudi from the GIT repository

The default checkout function (*defaultCheckout*) is equivalent to the code used in the old system, but simpler. If an explicit version is specified for a project, it is the version that is checked out, while the special version *HEAD* triggers the checkout of the head version of the project, meaning the head version of each package in the project. The old system required also the special flag *headofeverything* to achieve the same result. It is not possible, in the new system, to perform a checkout of the head meant as a checkout of the head version of the container package of a project and the versions there declared of the other packages, but this feature was never really used. After the straight forward checkout of the project, the *overrides* field is used to fine tune the sources: for each package declared in the list of overrides, `getpack` is called to change the already checked out version of the package or to add it if not present, while when the requested version is *null* the directory of the package is removed (if present).

[2]Remember that the new configuration requires one file per slot.

[3]It is enough to use the function name for functions in the module implementing the checkout script, while functions accessible from other modules require the fully qualified name (i.e. `module.function`).

The *noCheckout* function is useful to just declared the version of a project to be used in the build of the other projects, but not to build it. The same functionality was achieved in the old system by either declaring an alternative dependency for a project or by declaring the project as *disabled*. In the new system, then, it is only possible to implicitly disable the build of a project by not checking it out.

The *specialGaudiCheckout* function has been used for testing the checkout of the Gaudi project from its GIT repository instead of using `getpack`. Because of its test nature, this function does not support the overrides of the packages. It must be noted that the old system did not provide a way to change the checkout procedure of a project without deep changes in the implementation the system, so the use of this simple test function demonstrates the flexibility of the new system.

When invoked, the script checks out all the projects in the directory `builds` under the current working directory, then it modifies the configuration files of the projects to synchronize the interdependencies with what is declared in the configuration. In particular, for the CMake configuration it modifies the top `CMakeLists.txt` file changing the version of the project and of the dependencies, while for the CMT configuration it modifies the file `project.cmt` for the dependencies and the file `requirements` of the container package to remove the explicit versions of the packages, which, anyway, are not needed after the checkout. Every modification applied to a file is recorded in patch file in the directory `sources` under the current working directory.

After having patched the configuration, the sources of each project are packed in individually in the `sources` directory.

The patch file and the archives of the sources have in the name a *build id* that can be specified on the command line (by default `<slot>.<YYYY-MM-DD>`).

Together with the improvements and simplifications, there a few known limitations with the new checkout procedure.

When a package is added or removed from a project, the requirements file of the container package is not correctly modified, but the common use cases are such that this correction is very rarely needed. In case of the addition, the build procedure visits the extra packages even if they are not declared in the container, causing problems only if the runtime requires some special environment variable defined in the requirements files of the new package (of course, it is not a problem if the extra package is used by another package in the project). The only valid use case for the removal of a package is when the the package was overridden from a used project and we need to build using the original version of the package, because a "real" removal of a package will probably need non trivial corrections in other packages.

Another limitation of the new system is that it is not possible to build in the same slot two different versions of a project, but, even if theoretically possible in the old system, this feature was never used or needed.

4.1.3 Preconditions Check

In the old system it was possible to wait for a special file to appear before starting the build. This feature was meant to allow chaining of nightly builds, so that we can have a slot that is built on the products of a slot in the LCG AA nightly builds.

To achieve the same result in the new system, a more generic and flexible mechanism has been devised: we can use the configuration field *preconditions* (the only one used in this step) to declare functions to be called before building the slot.

The old “wait for” functionality is obtained via the precondition function *waitForFile*, which accepts the arguments:

path the path to the file we have to wait for

timeout time before giving up waiting (`datetime.timedelta`, optional)

maxAge a file older than this are ignored (`datetime.timedelta`, optional)

For example:

```
...
"preconditions": [
   {"name": "waitForFile",
    "args": {
      "path": "${LCGNIGHTLIES}/dev3/${TODAY}/isDone-${CMTCONFIG}"
    }}
],
...
```

This function waits until the requested file appears before returning successfully (*True*), but if the file does not appear within the timeout (by default 20 hours), the function returns a failure (*False*) and the build is not started.

In the future we can have other precondition functions with more complex logic, either returning immediately (e.g. if there is not enough disk space) or waiting.

4.1.4 Build and Test

Building and testing actions are bundled in one script because they are tightly connected: the tests of a project in the slot are run while other projects are built. The behavior of the build script can be tuned with documented command line options to allow easy and effective testing of the build system.

The configuration fields used in this step are *slot*, *projects* (only *name*, *version* and *dependencies*), *warning_exceptions*, *error_exceptions*, *USE_CMT* and *cmake_cache*.

To ensure a clean build, we first remove the content of the directory `builds` under the current working directory, then we unpack all the archives present in the `sources` directory (those produced during the checkout step). For each project found in the configuration and for which the sources are found (i.e. we ignore the projects that were not checked out) we create, from templates, a few configuration files and a script to be run via the CTest [5] (`ctest`) command (the build and test tool distributed with CMake), and, for the whole slot, we create a special file used to describe the structure of the slot to CDash.

The generated CTest script is the core of the build procedure. It is written such that it can build either CMake-based projects or CMT-based [6] ones (actually via the special Makefile that `getpack` adds to the project). With this script we can build and test a projects and submit the result to a CDash instance. The submission is optional, and we can also just build or just test.

Once the configuration is ready, the build wrapper script triggers CTest to build each project (optionally with a number of parallel processes, fixed with a command line option), one after the other in order of dependencies (as declared in the configuration[4]). Once a project is built, we first scan the build log recorded by CTest to produce summary files equivalent to those produced by the old Nightly Build System (see 5.2), then we pack the content of its `InstallArea` directory in an archive (dereferencing symbolic links) in the `builds` directory named after the project name, its version, a build id (like the one used in the checkout) and the platform (`$CMTCONFIG`). At this point we start a new subprocess in background to run the tests of the project while the next project in the slot is being built.

The the way the tests of a project are started depends on the type of the build (CMake or CMT), but in both cases we produce the same kind of summaries that were used by the old system. For CMake builds, the results of the tests are submitted to the CDash instance, but the details about all the QMTest tests are not visible there. In the near future we will implement an extension for QMTest to report the results in the XML format understood by CDash, so that we will have more meaningful details accessible through the CDash instance.

After all the builds are completed, the main script waits until the spawned subprocesses complete before exiting.

It is important to note that the archive of the build does not contain any artifact that has not been installed (copied or linked) to the `InstallArea` directory. This behavior is different from what is done when creating the standard LHCb distribution packages, and these new archives may be faulty. Of course, we can change the packaging to create archives strictly identical to those prepared in the release procedure, but the long term plan is to fix the rare cases where the artifacts are not installed[5].

4.1.5 Project Layout

As already mentioned, the Core Tools are hosted in a GIT repository. The projects features the following directories:

python contains the main Python module `LHCbNightlies2` with submodules for the shared code, the task specific functions and the tests, plus some templates

[4]The dependencies are not declared in the old configuration, but the order of appearance is used instead, so the automatic translation of the configuration introduces fake dependencies between the projects to preserve the expected order of build.

[5]A test to spot these cases is not available yet.

scripts contains the executable scripts, which are simply forwarding the calls to the main functions in the Python modules

docs documentation

In the top level directory there are two setup shell scripts (*sh* and *csh* variants) than can be sources to add the two main directories to `PYTHONPATH` and to `PATH`.

The tests are based on the `nose` testing framework [7] and can be run with:

```
cd python
nosetests -v --with-doctest
```

or with the simple wrapper `test.sh`.

The repository features two main branches: *master* and *dev*. The first one is the one used in production (automatically checked out by the Jenkins jobs), while the second is used for testing new features without interfering with the production jobs.

4.2 Jenkins Configuration

Jenkins is a very flexible and extensible application used to manage automated build and test jobs. Its features span from regular, on demand or commit triggered builds to complex build work-flows or just monitoring of tasks. Its plugin system allows to add functionalities like integration with different tools (e.g. Coverity or JIRA) and more reports from the builds.

Unfortunately our use case is so special that there is no plugin already available to support it, but, even if it could be implemented, we could achieve a good approximation with the right combination of some of the already existing plug-ins (the full list is reported in appendix A).

We shall not describe the installation and set up of the Jenkins server or the configuration of access and privileges because it is not relevant to the operation of the nightly builds. What is relevant, though, is the way we configure the cluster of build machines at our disposal. In the old system the build machines were configured to start routinely the builds (via cron jobs); in the new system it is the Jenkins instance that is aware of each build node (slave) and connects to them via `ssh` (using the `ssh` *Launch method* provided by the *Jenkins SSH Slaves Plugin*) with the credentials of the service account `lhcbsoft`. Each slave is configured with *Labels* set to the short OS id (for example *slc6* for *Scientific Linux CERN 6.x*). To avoid problems with the AFS token, we change the default *Availability* to take the node off line if idle (*Idle delay*: 20 minutes) and to connect if needed (*In demand delay*: 1 minute). Because of the particular configuration of the build machines, we have to set the *Remote FS root* to point to a directory on the dedicated disk (`/build/jenkins`). During the tests we did not have Java installed on the slaves (or the version was too old), so we installed the JRE[6] by hand on each node in the `jre` directory under the configured *Remote FS root* and declared it in the configuration (field

[6]Java Runtime Environment [8].

JavaPath in the advanced section of *Launch method*). The number of executors (*# of executors*) is usually set to the number of CPUs of the slave, but our builds and tests use on average two CPUs per job[7], so we set it to half the number of CPUs, except for the *master* node, that we use only for jobs that almost do not use CPU, where we use ~ 200 executors[8]. In the global configuration page (*Configure System*) the checkbox *Allow token macro processing*, under *Groovy*, must be ticked.

The jobs we configured for the nightly builds are divided in two categories: the workers and the slots. For a simpler monitoring the jobs of the two categories are grouped in two views.

4.2.1 The Workers

The workers are generic jobs that take care of the actual checkouts and builds of the slots.

Part of the configuration is common to all worker jobs. They, via the configuration *Discard Old Builds*, keep the details of the builds for 15 days and the artifacts for one week, which could be changed increasing the available disk space; of course the retention periods must be identical for all the jobs for consistency. Two parameters are common to all the workers:

slot string parameter defining the name of the slot

slot_build_id a numeric id (in a string parameter) common to all the jobs connected to the same build of a slot

In all cases we allow concurrent builds via the flag *Execute concurrent builds if necessary.* Using the *Jenkins GIT Plugin* we check out the Nightly Builds Core Tools from the URL mentioned in 4.1, to use the latest production version for their tasks, and we tick the *Clean after checkout* checkbox in the advanced section. Since each worker job will be used several times concurrently to work on different slots and platforms, to more easily identify the builds referring to a specific slot we use the *Build Name Setter Plugin* to change the name of the build from the default (a number) to the more useful `<slot>.<slot_build_id>`[9].

In order to reduce the load on the software repositories, the checkout of the code to be compiled is performed only once per slot by the job called *nightly-slot-checkout.* To ensure that the checkout is run on a slave that has the correct environment, we use the *Label Expression* field, under the section *Restrict where this project can be run*, with a value like `!master`. The builder section of the checkout job consists of a single shell script that,

[7]The average number of CPUs per job is not really predictable because we launch parallel builds using `distcc` [9, 10] and we run the tests in background, so we used the Unix command `time` to monitor the *real time* (world clock time) spent for a build versus the *user time* (CPU time).

[8]Some special Jenkins jobs do not occupy executors, like the parent job of a multi-configuration job, but it is not (jet) possible to configure a regular job as "flyweight" (the term used for the special jobs). When we will have a plugin to configure flyweight jobs, we can change that weird number of executors to a more reasonable value.

[9]Since the parameters cannot be use directly in the *Build Name* field, we use the fact that they are used to set environment variables, so the *Build Name* used is `${ENV,var="slot"}.${ENV,var="slot_build_id"}`.

essentially, calls the checkout script from the Core Tools (see 4.1.2) after having prepared the standard LHCb environment and, for testing, downloaded the old XML configuration into the directory `sources`. For the post-build actions we use *Archive the artifacts* to keep a copy of the content of the directory `sources` and we trigger a delete of the workspace after the build (*Jenkins Workspace Cleanup Plugin*) to save disk space.

Since each slot needs to be build on several platforms, but, possibly, only when some preconditions are met, we use a simple *multi-configuration* (or *matrix*) job called *nightly-slot-build-trigger* with two extra parameters:

platform a string parameter for a space-separated list of platform the slot must be built on

node a node parameter (provided by the *Node and Label Parameter Plugin*) used just to force the execution of the matrix sub-jobs on a specific node, the *master*

In the *Configuration Matrix* section we declare one *Dynamic Axis* (from the *Dynamic Axis Plugin*) named *platform* with the value takes from the environment variable *platforms* (automatically defined from the parameter with the same name). In the *Build Environment* section, we use the *Matrix Tie Parent Plugin* to ensure that not only the sub-jobs are run on the master, but the parent job is run there too. The build steps section is more complex then the one used for the checkout. First we copy the old XML configuration file from the artifacts of the checkout job that was started with the same slot name and slot build id, using the *Copy Artifact Plugin* and choosing the latest successful build of the project identified by the string `nightly-slot-checkout/slot=$slot,slot_build_id=$slot_build_id`. The second build step is a shell script that prepares the environment and calls the precondition check script from the Core Tools (see 4.1.3). The last step is to trigger the actual build jobs via the *Jenkins Parameterized Trigger Plugin*, so that the jobs *nightly-slot-build* (described later) are started with a *NodeLabel parameter* called *os_label* set to (as a single line[10])

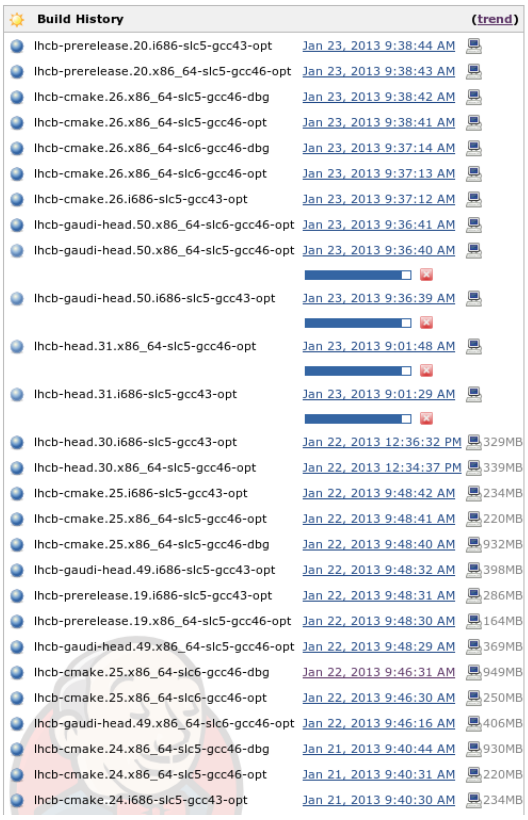
```
${GROOVY,script="import hudson.model.* ;
   def thr = Thread.currentThread() ;
   def build = thr.executable ;
   return build.getEnvironment().get('platform').split('-')[1]"}
```

and with *Predefined parameters* used to propagate the values of the parameters *slot*, *slot_build_id* and *platform*[11], then we *Block until the triggered projects finish their builds* and we, essentially, inherit the status of the triggered jobs. We do not need any special post-build action in this job.

The job triggered by *nightly-slot-build-trigger* is *nightly-slot-build* and it runs the heaviest part: the actual build and the tests. In addition to the common parameters it

[10]The *Groovy Plugin* allows execution of Groovy scripts [11] like in this case, and the *EnvInject Plugin* ensures that the environment variable *platform* is set from the axis.

[11]The *os_label* parameter is not passed in the same way as the others because the only way to trigger a job so that it is executed on a node different from the originating job is to use the special *NodeLabel parameter* type.



Build History (trend)

lhcb-prerelease.20.i686-slc5-gcc43-opt	Jan 23, 2013 9:38:44 AM	
lhcb-prerelease.20.x86_64-slc5-gcc46-opt	Jan 23, 2013 9:38:43 AM	
lhcb-cmake.26.x86_64-slc5-gcc46-dbg	Jan 23, 2013 9:38:42 AM	
lhcb-cmake.26.x86_64-slc5-gcc46-opt	Jan 23, 2013 9:38:41 AM	
lhcb-cmake.26.x86_64-slc6-gcc46-dbg	Jan 23, 2013 9:37:14 AM	
lhcb-cmake.26.x86_64-slc6-gcc46-opt	Jan 23, 2013 9:37:13 AM	
lhcb-cmake.26.i686-slc5-gcc43-opt	Jan 23, 2013 9:37:12 AM	
lhcb-gaudi-head.50.x86_64-slc6-gcc46-opt	Jan 23, 2013 9:36:41 AM	
lhcb-gaudi-head.50.x86_64-slc5-gcc46-opt	Jan 23, 2013 9:36:40 AM	
lhcb-gaudi-head.50.i686-slc5-gcc43-opt	Jan 23, 2013 9:36:39 AM	
lhcb-head.31.x86_64-slc5-gcc46-opt	Jan 23, 2013 9:01:48 AM	
lhcb-head.31.i686-slc5-gcc43-opt	Jan 23, 2013 9:01:29 AM	
lhcb-head.30.i686-slc5-gcc43-opt	Jan 22, 2013 12:36:32 PM	329MB
lhcb-head.30.x86_64-slc5-gcc46-opt	Jan 22, 2013 12:34:37 PM	339MB
lhcb-cmake.25.i686-slc5-gcc43-opt	Jan 22, 2013 9:48:42 AM	234MB
lhcb-cmake.25.x86_64-slc5-gcc46-opt	Jan 22, 2013 9:48:41 AM	220MB
lhcb-cmake.25.x86_64-slc5-gcc46-dbg	Jan 22, 2013 9:48:40 AM	932MB
lhcb-gaudi-head.49.i686-slc5-gcc43-opt	Jan 22, 2013 9:48:32 AM	398MB
lhcb-prerelease.19.i686-slc5-gcc43-opt	Jan 22, 2013 9:48:31 AM	286MB
lhcb-prerelease.19.x86_64-slc5-gcc46-opt	Jan 22, 2013 9:48:30 AM	164MB
lhcb-gaudi-head.49.x86_64-slc5-gcc46-opt	Jan 22, 2013 9:48:29 AM	369MB
lhcb-cmake.25.x86_64-slc6-gcc46-dbg	Jan 22, 2013 9:46:31 AM	949MB
lhcb-cmake.25.x86_64-slc6-gcc46-opt	Jan 22, 2013 9:46:30 AM	250MB
lhcb-gaudi-head.49.x86_64-slc6-gcc46-opt	Jan 22, 2013 9:46:16 AM	406MB
lhcb-cmake.24.x86_64-slc5-gcc46-dbg	Jan 21, 2013 9:40:44 AM	930MB
lhcb-cmake.24.x86_64-slc5-gcc46-opt	Jan 21, 2013 9:40:31 AM	220MB
lhcb-cmake.24.i686-slc5-gcc43-opt	Jan 21, 2013 9:40:30 AM	234MB

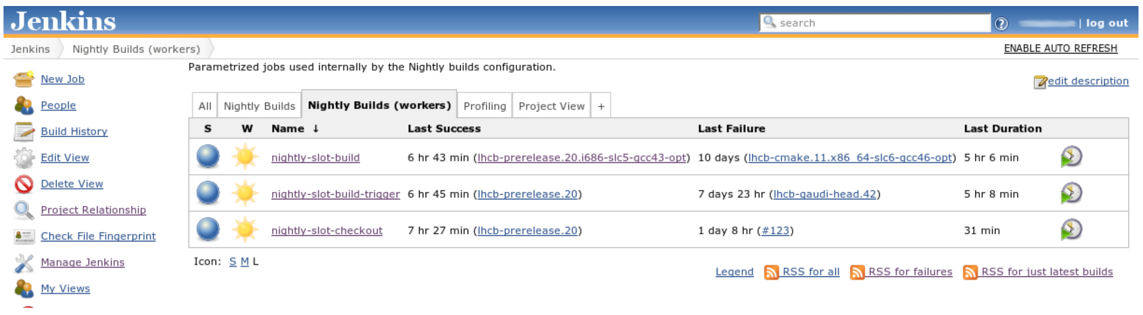
Figure 1: Jenkins: List of running and recently completed *nightly-slot-build* jobs.

accepts the string parameter *platform* declaring the platform id and a *Label* parameter called *os_label*[12]. Since this job will build the same slot on several different platforms, we appended `.${ENV,var="platform"}` to the *Build Name* used in the other jobs (Figure 1). In the build steps we copy all the artifacts from the latest successful build of the corresponding build of *nightly-slot-checkout* (selected as in the build trigger job), then we use a shell script to prepare the build environment and call the build script from Core Tools)[13] (see 4.1.4. In the post-build actions we archive the artifacts, in particular the archive files and the summaries found in the `build` directory.

The worker jobs are grouped and displayed in the view *Nightly Builds (workers)* (Figure 2), so that they can be easily monitored and not be confused with the slot jobs.

[12]A *Label* parameter is used by Jenkins to choose on which slave the job has to be run.

[13]To test the possible integration of the new nightly build system with the old summary page, we also copy to AFS summary files equivalent to the old ones that we generate during the build (see 5.2).



Parametrized jobs used internally by the Nightly builds configuration.

S	W	Name ↓	Last Success	Last Failure	Last Duration
●	☀	nightly-slot-build	6 hr 43 min (lhcb-prerelease.20.i686-slc5-gcc43-opt)	10 days (lhcb-cmake.11.x86_64-slc6-gcc46-opt)	5 hr 6 min
●	☀	nightly-slot-build-trigger	6 hr 45 min (lhcb-prerelease.20)	7 days 23 hr (lhcb-gaudi-head.42)	5 hr 8 min
●	☀	nightly-slot-checkout	7 hr 27 min (lhcb-prerelease.20)	1 day 8 hr (#123)	31 min

Figure 2: Jenkins: View of the worker jobs.

4.2.2 The Slots

To define and build the various slots we use dummy parameterized jobs, one per slot, named after the slot itself and where the only difference between them is the name and the default value of the single string parameter *platforms*[14], defining the list of platforms the slot is to be built on.

The configuration of these jobs is relatively simple[15]. The build details and artifacts retention policies are the same as the one used in the configuration of the workers (see 4.2.1), for consistency. The single string parameter *platforms* has already been mentioned and is used to define the list of platforms to build when manually triggering the build, but its default value is what is used for the regular builds, so, while in the old system the list of platforms to be built was stored in the configuration file, in the new system is part of the configuration of the slot job. Since these are dummy jobs not using CPU, we restrict their execution to the master node (*Restrict where this project can be run*). For these jobs we do not need any source code management.

The build of a slot is triggered via the *Build periodically* configuration box, which is set to start the builds between 00:00 and 01:00, using the special schedule `H 0 * * *`, similar to the specification of the Unix `cron` command with the special extension `H` that tells Jenkins to distribute the builds. The scheduling can be tuned to build only on some days as it was possible with some special (and less flexible) settings in the old configuration.

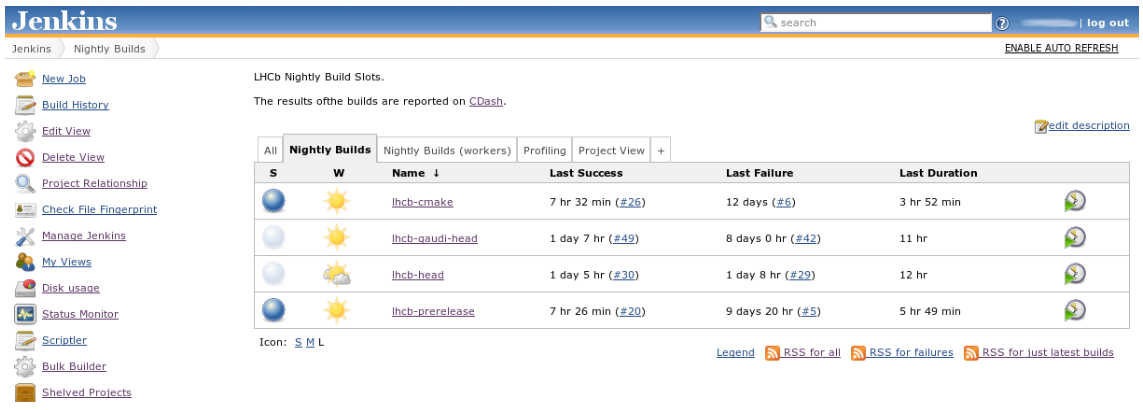
The build steps for the slots are two simple build triggers, the first one on *nightly-slot-checkout* and the second one on *nightly-slot-build-trigger*. In both cases we pass the following predefined parameters:

```
slot=${JOB_NAME}
slot_build_id=${BUILD_NUMBER}
```

and we wait for the completion of the triggered job, inheriting the status of the build.

[14]A string parameter is not very handy to specify the list of platforms and the Extended Choice Parameter Plugin could be used to improve usability, but it has not been tested yet.

[15]It could be further simplified by adding one more level of indirection (i.e. another parameterized worker job), but for the time being it doesn't seem necessary.



LHCb Nightly Build Slots.

The results ofthe builds are reported on CDash.

S	W	Name ↓	Last Success	Last Failure	Last Duration
●	☀	lhcb-cmake	7 hr 32 min (#26)	12 days (#6)	3 hr 52 min
○	☀	lhcb-gaudi-head	1 day 7 hr (#49)	8 days 0 hr (#42)	11 hr
○	⛅	lhcb-head	1 day 5 hr (#30)	1 day 8 hr (#29)	12 hr
●	☀	lhcb-prerelease	7 hr 26 min (#20)	9 days 20 hr (#5)	5 hr 49 min

Figure 3: Jenkins: View of the jobs representing the slots.

As for the workers, for easier monitoring, the slot jobs are grouped in a view called *Nightly Builds* (Figure 3).

4.2.3 Management Plugins

Not strictly needed to run the nightly builds, some plugins have been found very useful for the simplification of management tasks.

Bulk Builder start several jobs in one go, for example selecting all the jobs in a view, so that we can re-start all the slots with a few clicks

Jenkins Disk Usage Plugin show statistics on the disk space used by the various builds

Rebuilder restart a parameterized build reusing the same parameters, allowing us to restart the build of a single slot or platform without a new checkout

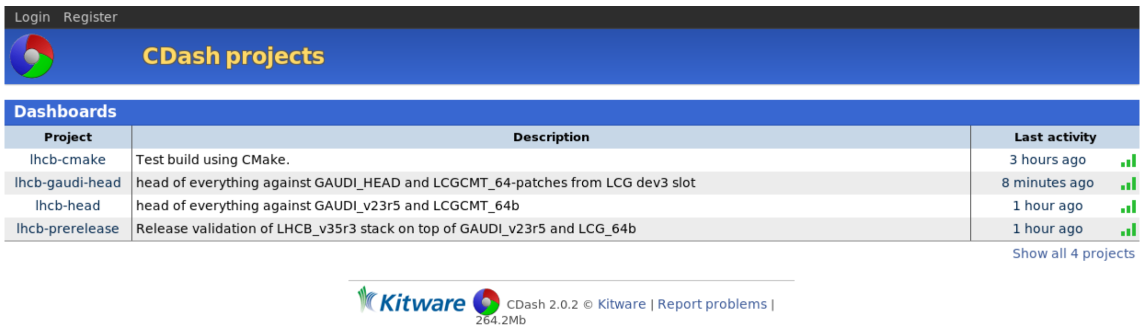
Recipe Plugin used to store in a file the configuration of a complex Jenkins setup (jobs, views, plugins) as a *recipe* and import it in another Jenkins instance, with it we can store the complete configuration of the nightly build system described in this note

Shelve Project Plugin archive projects for possible later use, instead of deleting them, which is useful to keep a back up copy of some old or temporary slots

5 Dashboard

It is extremely important for our developers and project managers to have a quick overview on the status of the nightly builds, and this view should be tunable so that each user can see only the informations relevant to her.

For the prototype of the new nightly builds system we considered two possibilities: CDash and the summary web page of the old system.


Login Register

CDash projects

Dashboards

Project	Description	Last activity
lhcb-cmake	Test build using CMake.	3 hours ago
lhcb-gaudi-head	head of everything against GAUDI_HEAD and LCGCMT_64-patches from LCG dev3 slot	8 minutes ago
lhcb-head	head of everything against GAUDI_v23r5 and LCGCMT_64b	1 hour ago
lhcb-prerelease	Release validation of LHCB_v35r3 stack on top of GAUDI_v23r5 and LCG_64b	1 hour ago

Show all 4 projects

Kitware CDash 2.0.2 © Kitware | Report problems | 264.2Mb

Figure 4: CDash: Initial page with the list of slots.

5.1 CDash

CDash is the dashboard solution developed for integration with the CTest tool bundled with CMake.

CTest can be used to build and test projects based on CMake as well as on custom tools. Of course, CMake projects are easier to handle than the others, but it has not been too difficult to develop a working script for CMT-based projects, as described in 4.1.4.

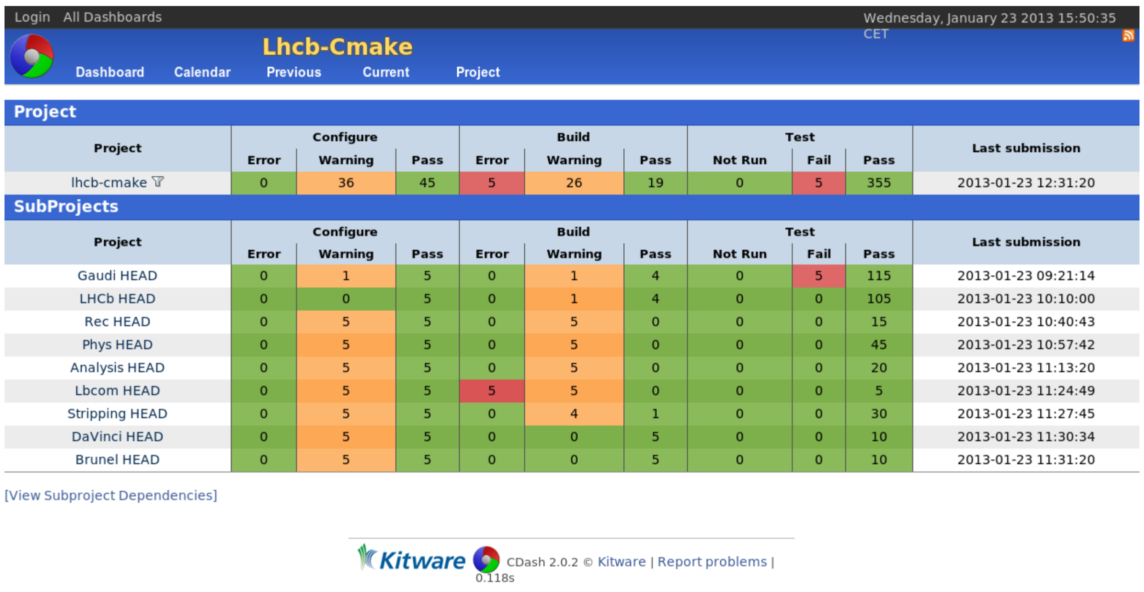
To try to reproduce a structure similar to the one we need, we declared each slot as a project in CDash and all the software projects in the slot as sub-projects [12,13] (using the combination of name and version as CDash sub-project names). The platform id string translates to the *build name* and we use the build slave hostname as *site*. In order to display enough informations, the CDash projects representing the slots must be configured to be public and display *labels* (special meta-data fields used to identify the builds of the sub-projects).

With this configuration, the first page we get when connecting to our CDash instance (currently `https://lbtestbuild.cern.ch/CDash`) contains the list of the declared slots (if a slot is not declared to CDash, it will not appear even if it has been built and the result pushed to the server) with a description, but without any information on the status of the build (Figure 4).

Clicking on one of the slots, we can access the overview page for that slot, where the number of successful or problematic builds is reported (Figure 5).

We can go deeper and see the details of the builds of project in the slot (Figure 6) or a summary for a platform (Figure 7).

It must be noted that the tests we run in our nightly builds use QMTest [14] (as mentioned in 4.1.4). When using CTest, we start a collection of QMTest tests as a single CTest test, so the numbers in the CDash views (Figure 6) are not correct, while, when we use CMT, we cannot even publish the results of the tests to CDash, so those fields are empty. The plan is to extend the output format of QMTest to produce result files in a format understood by CDash, so that we can publish the correct informations. In the long term we will also replace QMTest with CTest, but the extension of the output format is


Login All Dashboards

Wednesday, January 23 2013 15:50:35 CET

Lhcb-Cmake

Dashboard Calendar Previous Current Project

Project

Project	Configure			Build			Test			Last submission
	Error	Warning	Pass	Error	Warning	Pass	Not Run	Fail	Pass	
lhcb-cmake	0	36	45	5	26	19	0	5	355	2013-01-23 12:31:20

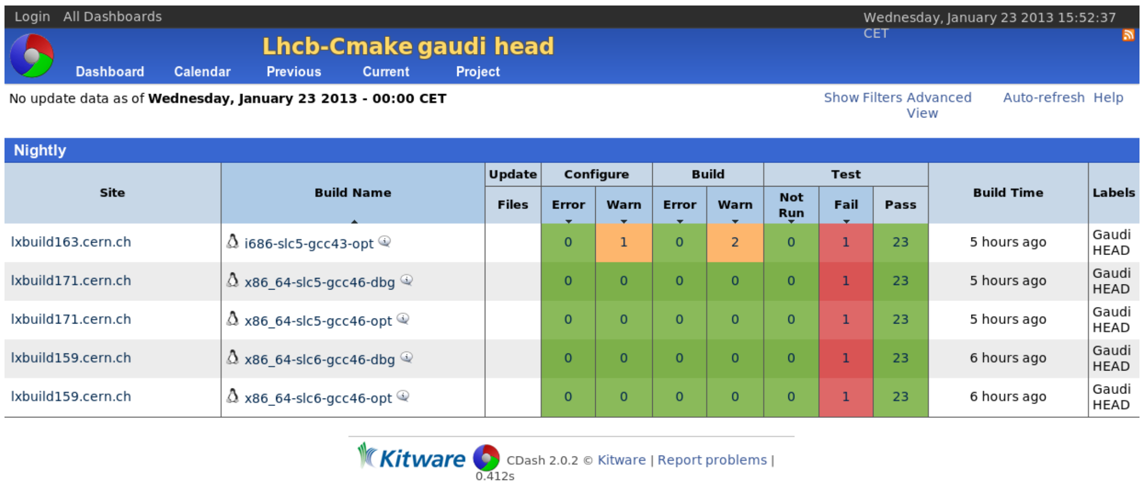
SubProjects

Project	Configure			Build			Test			Last submission
	Error	Warning	Pass	Error	Warning	Pass	Not Run	Fail	Pass	
Gaudi HEAD	0	1	5	0	1	4	0	5	115	2013-01-23 09:21:14
LHCb HEAD	0	0	5	0	1	4	0	0	105	2013-01-23 10:10:00
Rec HEAD	0	5	5	0	5	0	0	0	15	2013-01-23 10:40:43
Phys HEAD	0	5	5	0	5	0	0	0	45	2013-01-23 10:57:42
Analysis HEAD	0	5	5	0	5	0	0	0	20	2013-01-23 11:13:20
Lbcom HEAD	0	5	5	5	5	0	0	0	5	2013-01-23 11:24:49
Stripping HEAD	0	5	5	0	4	1	0	0	30	2013-01-23 11:27:45
DaVinci HEAD	0	5	5	0	0	5	0	0	10	2013-01-23 11:30:34
Brunel HEAD	0	5	5	0	0	5	0	0	10	2013-01-23 11:31:20

[View Subproject Dependencies]

Kitware CDash 2.0.2 © Kitware | Report problems | 0.118s

Figure 5: CDash: Overview of one slot.


Login All Dashboards

Wednesday, January 23 2013 15:52:37 CET

Lhcb-Cmake gaudi head

Dashboard Calendar Previous Current Project

No update data as of **Wednesday, January 23 2013 - 00:00 CET**

Show Filters Advanced View Auto-refresh Help

Nightly

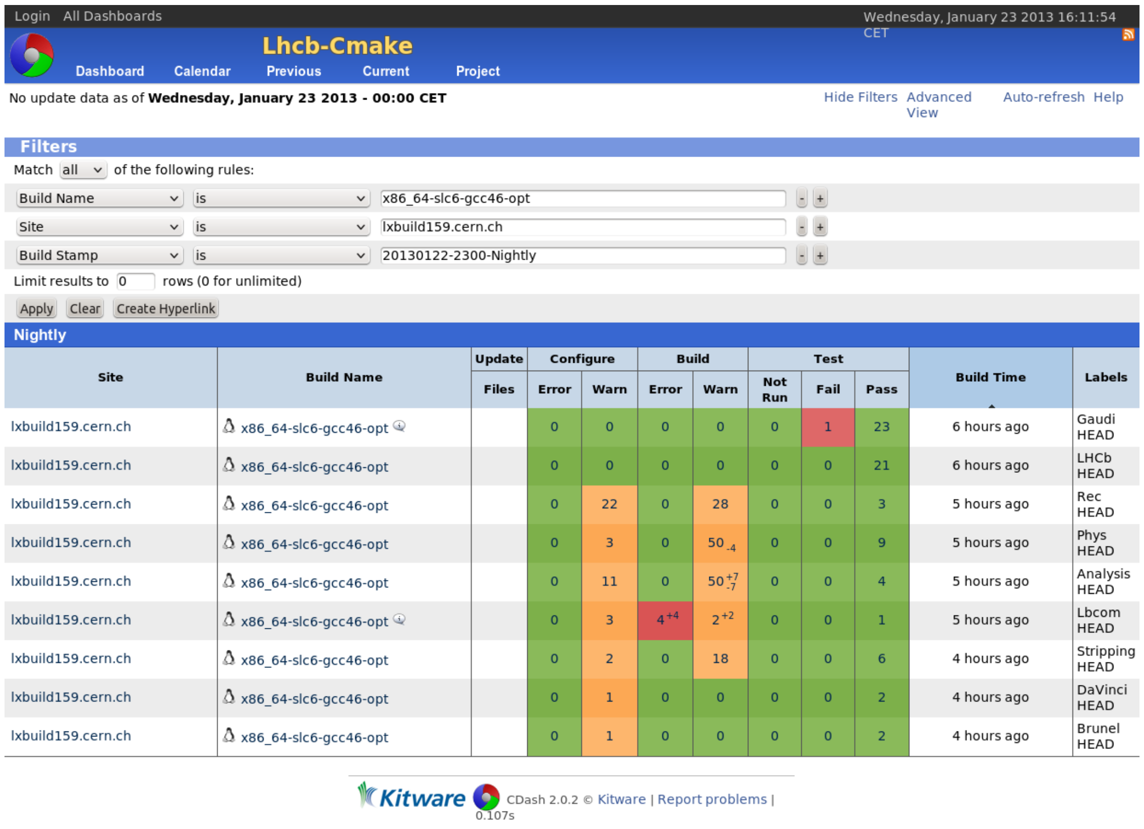
Site	Build Name	Update	Configure		Build		Test			Build Time	Labels
		Files	Error	Warn	Error	Warn	Not Run	Fail	Pass		
lxbuild163.cern.ch	i686-slc5-gcc43-opt		0	1	0	2	0	1	23	5 hours ago	Gaudi HEAD
lxbuild171.cern.ch	x86_64-slc5-gcc46-dbg		0	0	0	0	0	1	23	5 hours ago	Gaudi HEAD
lxbuild171.cern.ch	x86_64-slc5-gcc46-opt		0	0	0	0	0	1	23	5 hours ago	Gaudi HEAD
lxbuild159.cern.ch	x86_64-slc6-gcc46-dbg		0	0	0	0	0	1	23	6 hours ago	Gaudi HEAD
lxbuild159.cern.ch	x86_64-slc6-gcc46-opt		0	0	0	0	0	1	23	6 hours ago	Gaudi HEAD

Kitware CDash 2.0.2 © Kitware | Report problems | 0.412s

Figure 6: CDash: Detailed view of a project in one slot.

the only possibility in the short term.

The informations stored in the CDash database are enough for out purposes (apart from the results of the tests), but the layout is not optimal for our purposes. Since we build the same version of a project in several contexts, it is useful to see all its builds across the slots, but this is impossible with CDash. Moreover, the view in Figure 7 puts the accent on the wrong information (the site), while we give more importance to the



Login All Dashboards

Wednesday, January 23 2013 16:11:54 CET

Lhcb-Cmake

No update data as of **Wednesday, January 23 2013 - 00:00 CET**

Filters

Match all of the following rules:

Build Name	is	x86_64-slc6-gcc46-opt
Site	is	lxbuild159.cern.ch
Build Stamp	is	20130122-2300-Nightly

Limit results to 0 rows (0 for unlimited)

Nightly

Site	Build Name	Update	Configure		Build		Test			Build Time	Labels
		Files	Error	Warn	Error	Warn	Not Run	Fail	Pass		
lxbuild159.cern.ch	x86_64-slc6-gcc46-opt		0	0	0	0	0	1	23	6 hours ago	Gaudi HEAD
lxbuild159.cern.ch	x86_64-slc6-gcc46-opt		0	0	0	0	0	0	21	6 hours ago	LHCb HEAD
lxbuild159.cern.ch	x86_64-slc6-gcc46-opt		0	22	0	28	0	0	3	5 hours ago	Rec HEAD
lxbuild159.cern.ch	x86_64-slc6-gcc46-opt		0	3	0	50_{-4}	0	0	9	5 hours ago	Phys HEAD
lxbuild159.cern.ch	x86_64-slc6-gcc46-opt		0	11	0	50^{+7}_{-7}	0	0	4	5 hours ago	Analysis HEAD
lxbuild159.cern.ch	x86_64-slc6-gcc46-opt		0	3	4^{+4}	2^{+2}	0	0	1	5 hours ago	Lbcom HEAD
lxbuild159.cern.ch	x86_64-slc6-gcc46-opt		0	2	0	18	0	0	6	4 hours ago	Stripping HEAD
lxbuild159.cern.ch	x86_64-slc6-gcc46-opt		0	1	0	0	0	0	2	4 hours ago	DaVinci HEAD
lxbuild159.cern.ch	x86_64-slc6-gcc46-opt		0	1	0	0	0	0	2	4 hours ago	Brunel HEAD

Kitware CDash 2.0.2 © Kitware | Report problems | 0.107s

Figure 7: CDash: Overview of a platform in one slot.

project (label). We also observed some annoying bugs which might be fixed in a future version (the latest release is one year old). Anyway, since CDash is open source, we could extend it, more or less easily, to extract from the database the details we need and display them the way we like.

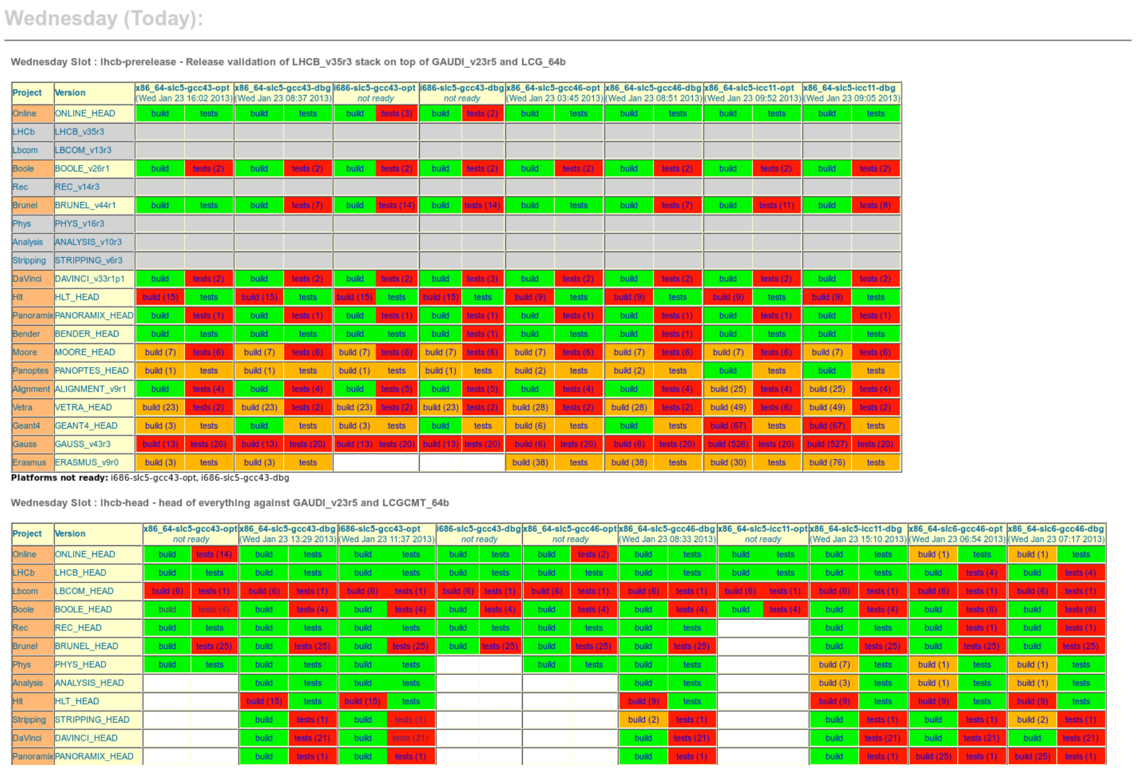
5.2 Old Summary Web Page

The summary web page of the old system was designed to convey all the needed informations in a single view (Figure 8), so its layout and infrastructure could be reuses. To evaluate the feasibility of this approach, we produce with the new system partial summary files for the build and tests that are compatible with the expectations of the old system.

Although possible, there are several downsides in this approach.

The parsing of the build and test log files is partially integrated in the build step of the old system and partially in the regular job that generates the full version of the summary page. To produce the data required by the regular job we need to rewrite from scratch the parsing code to extract it from the old build step.

The technology used for the summary page is obsolete: a static HTML file produced



Wednesday (Today):

Wednesday Slot : lhcb-prerelease - Release validation of LHCB_v35r3 stack on top of GAUDI_v23r5 and LCG_64b

Project	Version	x86_64-slc5-gcc43-opt		x86_64-slc5-gcc43-dbg		i686-slc5-gcc43-opt		i686-slc5-gcc43-dbg		x86_64-slc5-gcc46-opt		x86_64-slc5-gcc46-dbg		x86_64-slc5-icc11-opt		x86_64-slc5-icc11-dbg	
		(Wed Jan 23 16:02 2013)		(Wed Jan 23 08:37 2013)		*not ready*		*not ready*		(Wed Jan 23 03:45 2013)		(Wed Jan 23 08:51 2013)		(Wed Jan 23 09:52 2013)		(Wed Jan 23 09:05 2013)	
Online	ONLINE_HEAD	build	tests	build	tests	build	tests (3)	build	tests (2)	build	tests	build	tests	build	tests	build	tests
LHCb	LHCB_v35r3																
Lbcom	LBCOM_v13r3																
Boole	BOOLE_v26r1	build	tests (2)	build	tests (2)	build	tests (2)	build	tests (2)	build	tests (2)	build	tests (2)	build	tests (2)	build	tests (2)
Rec	REC_v14r3																
Brunel	BRUNEL_v44r1	build	tests	build	tests (7)	build	tests (14)	build	tests (14)	build	tests	build	tests (7)	build	tests (11)	build	tests (8)
Phys	PHYS_v16r3																
Analysis	ANALYSIS_v10r3																
Stripping	STRIPPING_v6r3																
DaVinci	DAVINCI_v33r1p1	build	tests (2)	build	tests (2)	build	tests (2)	build	tests (3)	build	tests (2)	build	tests (2)	build	tests (2)	build	tests (2)
Hlt	HLT_HEAD	build (15)	tests	build (15)	tests	build (15)	tests	build (15)	tests	build (9)	tests	build (9)	tests	build (9)	tests	build (9)	tests
Panoramix	PANORAMIX_HEAD	build	tests (1)	build	tests (1)	build	tests (1)	build	tests (1)	build	tests (1)	build	tests (1)	build	tests (1)	build	tests (1)
Bender	BENDER_HEAD	build	tests	build	tests	build	tests	build	tests (1)	build	tests	build	tests (1)	build	tests	build	tests
Moore	MOORE_HEAD	build (7)	tests (6)	build (7)	tests (6)	build (7)	tests (6)	build (7)	tests (6)	build (7)	tests (6)	build (7)	tests (6)	build (7)	tests (6)	build (7)	tests (6)
Panoptes	PANOPTES_HEAD	build (1)	tests	build (1)	tests	build (1)	tests	build (1)	tests	build (2)	tests	build (2)	tests	build	tests	build	tests
Alignment	ALIGNMENT_v9r1	build	tests (4)	build	tests (4)	build	tests (5)	build	tests (5)	build	tests (4)	build	tests (4)	build (25)	tests (4)	build (25)	tests (4)
Vetra	VETRA_HEAD	build (23)	tests (2)	build (23)	tests (2)	build (23)	tests (2)	build (23)	tests (2)	build (28)	tests (2)	build (28)	tests (2)	build (49)	tests (6)	build (49)	tests (2)
Geant4	GEANT4_HEAD	build (3)	tests	build	tests	build (3)	tests	build	tests	build (6)	tests	build	tests	build (67)	tests	build (67)	tests
Gauss	GAUSS_v43r3	build (13)	tests (20)	build (13)	tests (20)	build (13)	tests (20)	build (13)	tests (20)	build (6)	tests (20)	build (6)	tests (20)	build (526)	tests (20)	build (527)	tests (20)
Erasmus	ERASMUS_v9r0	build (3)	tests	build (3)	tests					build (38)	tests	build (38)	tests	build (30)	tests	build (76)	tests

Platforms not ready: i686-slc5-gcc43-opt, i686-slc5-gcc43-dbg

Wednesday Slot : lhcb-head - head of everything against GAUDI_v23r5 and LCGCMT_64b

Project	Version	x86_64-slc5-gcc43-opt		x86_64-slc5-gcc43-dbg		i686-slc5-gcc43-opt		i686-slc5-gcc43-dbg		x86_64-slc5-gcc46-opt		x86_64-slc5-gcc46-dbg		x86_64-slc5-icc11-opt		x86_64-slc5-icc11-dbg		x86_64-slc6-gcc46-opt		x86_64-slc6-gcc46-dbg	
		not ready		(Wed Jan 23 13:29 2013)		(Wed Jan 23 11:37 2013)		*not ready*		*not ready*		(Wed Jan 23 08:33 2013)		*not ready*		(Wed Jan 23 15:10 2013)		(Wed Jan 23 06:54 2013)		(Wed Jan 23 07:17 2013)	
Online	ONLINE_HEAD	build	tests (14)	build	tests	build	tests	build	tests	build	tests (2)	build	tests	build	tests	build	tests	build (1)	tests	build (1)	tests
LHCb	LHCB_HEAD	build	tests	build	tests	build	tests	build	tests	build	tests	build	tests	build	tests	build	tests	build	tests (4)	build	tests (4)
Lbcom	LBCOM_HEAD	build (6)	tests (1)	build (6)	tests (1)	build (6)	tests (1)	build (6)	tests (1)	build (6)	tests (1)	build (6)	tests (1)	build (6)	tests (1)	build (6)	tests (1)	build (6)	tests (1)	build (6)	tests (1)
Boole	BOOLE_HEAD	build	tests (4)	build	tests (4)	build	tests (4)	build	tests (4)	build	tests (4)	build	tests (4)	build	tests (4)	build	tests (4)	build	tests (6)	build	tests (6)
Rec	REC_HEAD	build	tests	build	tests	build	tests	build	tests	build	tests	build	tests			build	tests	build	tests (1)	build	tests (1)
Brunel	BRUNEL_HEAD	build	tests (25)	build	tests (25)	build	tests (25)	build	tests (25)	build	tests (25)	build	tests (25)			build	tests (25)	build	tests (25)	build	tests (25)
Phys	PHYS_HEAD	build	tests	build	tests	build	tests			build	tests	build	tests			build (7)	tests	build (1)	tests	build (1)	tests
Analysis	ANALYSIS_HEAD			build	tests	build	tests					build	tests			build (3)	tests	build (1)	tests	build (1)	tests
Hlt	HLT_HEAD			build (15)	tests	build (15)	tests					build (9)	tests			build (9)	tests	build (9)	tests	build (9)	tests
Stripping	STRIPPING_HEAD			build	tests (1)	build	tests (1)					build (2)	tests (1)			build	tests (1)	build	tests (1)	build (2)	tests (1)
DaVinci	DAVINCI_HEAD			build	tests (21)	build	tests (21)					build	tests (21)			build	tests (21)	build	tests (21)	build	tests (21)
Panoramix	PANORAMIX_HEAD			build	tests (1)	build	tests (1)					build	tests (1)			build	tests (1)	build (25)	tests (1)	build (25)	tests (1)

Figure 8: Old Summary Web Page: Part of the summary tables.

by a job run every 15 minutes, which is then filtered by a CGI script. A more modern approach would involve dynamic content (currently achieved by forcing a reload of the full page) and an independent exchange format, such as XML, with the possibility of a layout usable on small screens (smart phones).

The amount of work required to reproduce completely the old summary files will be probably spent better in the extension of CDash or Jenkins, or in the development of a completely new dashboard.

6 Conclusions

We produced a working prototype of a new Nightly Build System based on already available tools reducing the customization to the minimum. The new build system demonstrated to be quite stable and reliable.

The current prototype is still missing a dashboard and the integration of QMTest with CTest (which is anyway needed as part of the migration our build system to CMake). We evaluated two possibilities for the implementation of the dashboard, but further studies and developments are needed to have a final version.

Appendix

A List of Required Jenkins Plug-Ins

Here is summarized the list of plug-ins that are required to configure Jenkins to run the nightly builds

- Build Name Setter Plugin
- Copy Artifact Plugin
- Dynamic Axis Plugin
- EnvInject Plugin
- Groovy Plugin
- Jenkins GIT Plugin
- Jenkins Parameterized Trigger Plugin
- Jenkins SSH Slaves Plugin
- Jenkins Workspace Cleanup Plugin
- Matrix Tie Parent Plugin
- Node and Label Parameter Plugin

The following list contains the plug-ins not needed for the jobs configuration, but useful for management tasks

- Bulk Builder
- Jenkins Disk Usage Plugin
- Rebuilder
- Recipe Plugin
- Shelve Project Plugin

References

[1] Karol Kruzelecki, Stefan Roiser, and Hubert Degaudenzi. The nightly build and test system for LCG AA and LHCb software. *J.Phys.Conf.Ser.*, 219:042042, 2010. doi:10.1088/1742-6596/219/4/042042.

[2] Jenkins CI [online]. URL: http://jenkins-ci.org/.

[3] GIT Distributed Version Control System [online]. URL: http://git-scm.com/.

[4] CDash - an open source, web-based software testing server [online]. URL: http://www.cdash.org/.

[5] CMake - Cross Platform Make [online]. URL: http://www.cmake.org/.

[6] CMT Home Page [online]. URL: http://www.cmtsite.org/.

[7] nose, a Python testing framework [online]. URL: https://nose.readthedocs.org.

[8] Java [online]. URL: http://java.com.

[9] distcc: a fast, free distributed C/C++ compiler [online]. URL: http://code.google.com/p/distcc/.

[10] CERN distcc Pilot Service [online]. URL: https://twiki.cern.ch/twiki/bin/view/LinuxSupport/DistccPilotService.

[11] Groovy scripting language [online]. URL: http://groovy.codehaus.org/.

[12] Bill Hoffman Ken Martin. *Mastering CMake*. Kitware, Inc., 2010.

[13] CDash Subprojects [online]. URL: http://www.kitware.com/media/html/CDashSubprojects.html.

[14] Jeffrey Oldham Mark Mitchell and Greg Wilson. QMTest: A Software Testing Tool. In *Tenth International Python Conference*, 2002. URL: http://www.python.org/workshops/2002-02/papers/01/index.htm.